

DOCUMENT RESUME

ED 332 660

IR 014 131

AUTHOR O'Malley, Claire E.
TITLE Helping Users Help Themselves.
INSTITUTION Open Univ., Walton, Bletchley, Bucks (England). Inst. of Educational Technology.
PUB DATE 86
NOTE 23p.; In: Norman, D. A. and Draper, S. W., Eds. User Centered Systems Design. Hillsdale, New Jersey, L. Erlbaum Associates, 1986. p.377-398. Figures contain very small type.
PUB TYPE Viewpoints (Opinion/Position Papers, Essays, etc.) (120) -- Reports - Evaluative/Feasibility (142)
EDRS PRICE MF01/PC01 Plus Postage.
DESCRIPTORS *Computer System Design; *Design Requirements; Foreign Countries; Guides; Individualized Instruction; *Information Seeking; *Menu Driven Software; *Questioning Techniques; *User Needs (Information)
IDENTIFIERS England

ABSTRACT

This discussion of the design of user-initiated help systems in computers focuses on the information that users actively seek to help them with their tasks, with emphasis on how to help users ask the questions that will bridge the gap between the initial internal (mental) form of the query and their information need as expressed by the system. Ways to help users ask questions are suggested, and examples are provided of information structuring on the Ceefax teletext system and the UNIX and Rabbit computer systems. The process of formulating a question is discussed, including the different types of questions that can crop up when going from an initial vague question to the final answer. It is argued that, although several different methods of getting help should be made available to the user, it is important that all of these systems form a well-integrated whole that enables the user to go from each subsystem to the next in a smooth and uninterrupted fashion. The need for dynamic structures that will enable users to structure or restructure the information to suit their own purposes is then discussed and "hypertext" is suggested as one method of providing such a structure. It is concluded that current systems tend to handle internally generated questions with natural language or database-query kinds of questions, and externally generated questions with "direct manipulation" interfaces. Designers are urged to consider both types of information gathering mechanisms more systematically for both kinds of questions. Seven figures, and eight footnotes are included. (EW)

* Reproductions supplied by EDRS are the best that can be made *
* from the original document. *

U.S. DEPARTMENT OF EDUCATION
Office of Educational Research and Improvement
EDUCATIONAL RESOURCES INFORMATION
CENTER (ERIC)

☐ This document has been reproduced as
received from the person or organization
originating it.

☒ Minor changes have been made to improve
reproduction quality.

• Points of view or opinions stated in this docu-
ment do not necessarily represent official
OERI position or policy.

CHAPTER 18

Helping Users Help Themselves

"PERMISSION TO REPRODUCE THIS
MATERIAL HAS BEEN GRANTED BY

E.S. Henderson

CLAIRE E. O'MALLEY

TO THE EDUCATIONAL RESOURCES
INFORMATION CENTER (ERIC)."

This chapter is motivated by two observations, or rather by two views of one observation: First, that users prefer to consult other people rather than to use the manuals and other types of help provided for them; second, that the people that users consult may be characterized not simply by how much they know, but also by their ability to get access to the relevant information.

The introduction to this section points out that the "information flow" perspective means paying attention to a *variety* of types of information that must reach users during the execution of their tasks—from such low-level feedback as the echoing of characters on the screen to the higher-level information necessary to plan the next activity. This perspective involves considering not just what information users actively seek, but also what they pick up by chance (Owen, Chapter 17); it involves considering not just the information that can be obtained from the computer system and the "official" documentation, but also what help can be obtained from other people (Bannon, Chapter 19).

In this chapter I focus on the information that users *actively seek*, rather than what they might be taught, or what they might pick up by chance. The situation is one in which a question has arisen, and an answer is sought: Questions first, answers later. The emphasis, however, is not so much on the issue of how to help users find information so much as on how to help users *ask the question* in the first place.

BEST COPY AVAILABLE

ED 332 660

IR 014 131

USERS HELPING EACH OTHER

Despite the variety of forms of computer help and documentation that are available, it is widely observed that users tend not to use them but prefer to consult the "local expert" or other users.¹ There are several possible reasons why users give up on manuals or help systems. Common complaints include:

- "I can't find the information"
- "There's too much information to wade through"
- "I don't have the time to spend searching"
- "I just want to check up on some detail"

Users want to be able to describe the "symptoms" of their problem and be told what to do, but they aren't able to because they often don't know how to describe the symptoms in the terms that can be found in "the book." In the case of computer manuals, users have the task of inferring the nature of the "illness" from a description of the functioning of a healthy "organism." Just as they would go to a doctor who can perform a diagnosis and make a prescription rather than looking up their symptoms in a medical handbook, users go to other people because they are better able to "diagnose" the problem and prescribe a solution. Taking this analogy further: Just as the doctor often doesn't cure people's illnesses directly, but tells them where to get the medication that will cure them, the computer consultant doesn't necessarily solve users' problems for them but instead tells them where to get the information that will help them solve the problem. The "expert," in both these cases, acts as an interpreter between "users" and "the system."

When users get help from other people, it is not always the case that the other person knows the solution. Often it is more the case that the other person knows *where to find the information* that will answer the

¹ Cf. Lang, Auld, and Lang, 1982; Lang, Lang, and Auld, 1981; Scharer, 1983.

user's question. One of the characteristics of the "local expert" in a user community is an ability to use the documentation and manuals. This is just as likely to be the reason why such people are frequently consulted, rather than the fact that they "know more" than other users.

This simple observation bears closer analysis: Users may go to the "expert" because they are unfamiliar with the way the available information is organized, and it is less effort to ask someone who is familiar with it. But what does this really mean? Knowing something about the way information is organized means being able to ask the right question to get at the required information. So another way of articulating this observation is that "experts" are able to point users towards the information they need because they are able to formulate the question better than users can.

Users have trouble finding the answer themselves because there is usually a huge gap between the initial internal (mental) form of the query and the information they need, as expressed by the system. Other people are often better at closing parts of that gap, by helping users translate their intentions into specific questions that can then be mapped onto the form in which the information is presented and organized in the system or the documentation.

HELPING USERS ASK QUESTIONS

How could you help users find the answers when they don't even know the questions to ask?

1. You could present some likely answers so that they could recognize what they want (i.e., you can give them the answers so that they can find the questions: see Owen, Chapter 17). For example if you ask travel agents to suggest a "good" holiday, they may try asking a few questions but will often end up dumping some brochures in front of you, waiting for you to recognize a "good holiday."
2. Rather than presenting the information in an unorganized way, you could help users select what they need by giving them an overview of the organization (structure) of the information. Figure 18.1 shows some examples of ways of structuring information.
3. You could adopt a middle path and present examples, then allow users to refine the examples in some way (e.g., by altering certain properties, or by criticizing certain aspects of the

A

P100 CEEFAX 100 Fri 30 Jan 16 09/57

BBC CEEFAX	
NEWS HEADLINES 101	FOR THE DEAF 169
NEWS IN DETAIL 102-119	SUBTITLES 170
NEWS FLASH 150	
NEWS INDEX 190	TV/RADIO 171-9
NEWSREEL 199	TV CHOICE 177
	TOMORROW 178
FINANCE HEADLINES 120	WEATHER/TRAVEL INDEX 180
FINANCIAL NEWS 121-139	GENERAL INDEX 193-5
SPORT HEADLINES 140	
SPORT PAGES 141-159	
FOOD GUIDE 161	
FARMING 160	
VALENTINE MESSAGES 201 (SEC 2)	

Index to the BBC's Ceefax Teletext System

B

```

HELP PRINT
PRINT
  Outlines one or more files for printing, either on a default system printer or on a specified device
  Format
    PRINT file-spec[. ]
  Additional information available:
  Parameters Qualifiers
  /AFTER=absolute-time /CHARACTERISTICS=[c[. ] ] /HOLD
  /DEVICE=device-name[ ] /FORMS=type /NOIDENTIFY
  /NOHOLD(D) /IDENTIFY(D) /LOWERCASE /NOLOWERCASE(F) /NAME=job-name
  /JOB_COUNT=n(D=1) /QUEUE=queue-name[ ] /BURST /NONBURST
  /PRIORITY=n /DELETE /NODELETE(D) /FEED(D) /NOFEED
  /FLAG,PAGE /NOFLAG,PAGE /HEADER /NOHEADER /PAGE_COUNT=n
  /SPACE[~n]

```

Print Help on the VAX/VMS

C

PERMUTED INDEX

pr: print file	pr(1)
print: pr to the line printer	print(1)
monitor, monstartup, moncontrol:	prepare execution profile
monitor, monstartup, moncontrol:	prepare execution profile
monitor, monstartup, moncontrol:	prepare execution profile
color: Alter scroll output for CRT	previewing
types:	primitive system data types
cat: concatenate and	print
lpr: off line	print
fortune:	print a random, hopefully interesting, oodge
date:	print and set the date
cal:	print calendar
hashstat:	print command hashing statistics
jobs:	print current job list
whoami:	print effective current user id
ps:	print file
lpr:	print Fortran file
history:	print history event list

UNIX 4.2 BSD Manual

FIGURE 18.1. Examples of ways of structuring information. (A) The British Broadcasting Corporation's *Ceefax* ("see facts") teletext system enables the viewer to choose certain pages (screenfuls) of information to display, by entering a page number via a remote control unit. (B) On the VAX/VMS system, a user wanting information on the print command types the command HELP followed by the argument PRINT. The system presents a brief explanation of the print command, followed by a list of possible parameters that can be specified along with it. More detailed assistance is available on each of these parameters by typing HELP PRINT/ followed by the parameter qualifier, e.g., HELP PRINT/IDENTIFY. (C) The permuted index to the UNIX manual allows the user to find manual entries for commands in several ways.

example that they don't like). This is the approach taken by the Rabbit system,² for example, and combines aspects of the first solution (present the answers) with aspects of the second (structure the information). Figure 18.2 illustrates how Rabbit does this.

However, the problem with solution 1 (present the answers) is that users are faced with the very problems they have with most existing help systems and manuals—there is too much information to wade through. The problem with solution 2, and to some extent, solution 3, is that one kind of organization of information will not necessarily work for all types of problem. Users' tasks more often than not cut across the boundaries forming the organization of the information.

For example, on our UNIX system, if you wanted to print a document with tables and equations on the laser printer, you might have to type the command line:

```
tbl filename | eqn -s8 -fl | ltruff -micsl
```

This single command-line contains three separate programs with their various associated flags or options. In order to get help for this task you would have to access information via the online manual from three different entries for the programs tbl, eqn, and ltruff—and you would have to spend some time on each query to find the relevant information, assuming you knew the names of the appropriate commands.

The point is that you often have to break down a single task into several components, and if help is required in subsequently constructing the command line, it has to be obtained for each of the components separately. The documentation is structured at the level of the system modules (commands) and not at the level of the task, nor at the level of a legitimate executable command line. This is not a gripe about one particular system: It is a problem for any system that offers users the power of combining units.

² Tou, Williams, Fikes, Henderson, and Malone, 1982.

Main Description	Description
	--Attributes of the query-- Entity Product Computer manufacturer: Apple, or Xerox --Attributes of the query--
Example --Attributes of Apple-II-plus-- Entity Product ComputerRelatedProduct OEM-Product RetailProduct Computer name: --- cost: \$1400, \$2,040 <u>manufacturer: Apple</u> --Attributes of Apple-II-plus-- --Information about Apple-II-plus--	Apple Atari Cromemco Xerox Intel Motorola Zilog Tandy Sharp Sony require any require all prohibit
Previous Description	28 Matching Examples
Entity Product ComputerRelatedProduct	Apple-II-plus Atari-400 Atari-800 Cromemco-Z-80 Star-8011 Star-8012 6502 68000 6802 6800 6805a Denelson

FIGURE 18.2. The Rabbit system allows the user to access information by reformulation of examples. In this particular example the user wants to find a computer with a bit-map display and a certain type of processor. An image of an example in the database matching the current partial description (*entity product*) appears in the example pane. The user then specifies that the manufacturer be either Xerox or Apple. The process of "critiquing" the example instance continues until the user is satisfied with the current partial description. Then another example is retrieved based on the new description, and so on until the desired item is found. (Example provided by M. Williams, personal communication.)

Formulating Questions Is a Multistep Process

There are two common types of problems that arise in formulating questions, both of which are often specifically addressed in existing help systems: First, knowing the name of the command to type in order to get help; second, knowing the name of the arguments to give.

For example, in UNIX, in order to ask the question "*how do I search for a line in a file?*" I have to formulate the question as "*man grep.*" Even if I know that the command for getting help is *man*, I still may not know which command I should ask about. A simple way of handling this problem is with keyword search facilities, such as the program on UNIX called *apropos*, (assuming I know its name and, indeed, of its existence). For instance, with our local version I could type "*apropos search for line in file*" and I would be given a list of possible commands, including the one that is needed:

grep, egrep, fgrep (1) — search a file for a pattern.

This is an example of the "bootstrapping" problem discussed in the section introduction. In order for users to ask for help they must already know about the help commands.

There are other kinds of attempts to take care of both of these problems. For example, the use of menus obviates the need to know how to ask for help by showing the user what information is available. This is a small-scale application of the "*answers first, then questions,*" approach within the framework of the "*questions first*" paradigm which "patches" the problem that users don't know how to ask questions. Of course, menus (and their pictorial counterparts icons) have their own set of difficulties, e.g., a complex hierarchical structure.

However, users often go to other people rather than consult the "official" sources of help because other people can help them ask the right question. This is more complicated than the simple case of the other person telling the user what command to type to find information. In other words, "asking the right question" goes beyond the problem of using the right language or terminology. The user often needs help in actually *formulating* the question, that is, in specifying it to a detailed enough level. Moreover, this process often involves several steps, not just one. It can be multistep in at least two senses: going

from the general to the specific (internally generated questions) and diagnosing problems (externally generated questions).

Going from the general to the specific: Internally generated questions. There may be several stages to go through from the first general question to the kind of specific question that can be asked of the system or documentation. Take an example where users have some general intention, e.g., they are trying to figure out how to format a letter for printing. They have some idea in mind of the eventual product, and they know how to create the actual text (i.e., how to edit a file), but not how to specify how they want it formatted. In order to get help in realizing their intention they have to translate it into specific questions so that they can make use of the help facilities to find the information they need. The translation process may involve a one-to-one or a one-to-many mapping—that is, one question may map straight onto another (e.g., "how do I search for a line in a file?" translates into "man grep"), or several questions may be involved (e.g., "how do I print a file on the laser printer" translates into "man tbl," "man eqn," "man ltrff," and so on).

Norman points out in Chapter 3 that there are several stages in going from very general intentions to the level of actually executing commands on the system. The complexity involved in the planning process has also been highlighted by several other studies (e.g., Moran's "external-to-internal-task mapping" model, 1983; "planning nets," Riley & O'Malley, 1985). An implicit corollary of these analyses is that users also need to be supported in formulating and asking questions of the help system that correspond to each of these stages. In other words, each step potentially involves one or more questions to be formulated and asked.³ Such support is needed particularly where questions are generated "internally"—that is, where they are generated from some plan that the user has not yet made explicit.

For example, Figure 18.3A illustrates what a typical unformatted letter might look like just after the user has typed it in. Figure 18.3B shows what might be desired as the final formatted product. Figure 18.3C shows the formatting commands that would actually have to be inserted in the file on our system. Finally, the user would have to type the command line *lpr file*, in order to have the letter printed on the laserprinter. In this case it is clear that it would be rather unhelpful if the person who was consulted simply told the user to type *lpr*. For

³ In fact, you could look at this point in another way: Observation of the questions generated naturally at each stage might serve to validate these analyses.

example, let's say the user's most general level question is: "*how do I get a formatted printout of a letter?*" Assuming the user actually knows that they have to put formatting commands in the file in order to get a formatted version printed, the next specified question might be "*how do I get a letter heading?*" or "*how do I get the date printed on the right hand side?*" "*how do I indent the paragraphs?*" and so forth. What is needed is to go through a whole series of questions and answers in order to determine what is required for accomplishing the whole task.

Diagnosing problems: Externally driven questions. Let's now look at a slightly different example concerning the problem of formulating questions. Figure 18.4 is adapted from the example given by Norman (see Chapter 3). This figure illustrates that, although there may be a one-to-one mapping between the initial diagnosis of the problem (e.g., "*I need to make this paper look better*") and the intention ("*I have to change .pp to .sp*"), several steps need to be taken to determine the specific problem. For example, the user has used the right formatting commands in the file, at least as far as the system is concerned, but then finds that the formatted version of the letter is not really what was wanted. However, there may be problems in figuring out exactly what is wrong. All that's known is that the letter "doesn't look right."

Since there was no error as far as the system is concerned (the formatting commands are "correct"), the user somehow has to convey the fact that it wasn't what was wanted—which is difficult to do on most systems. For example, if I went to another person for help, I could point to a printout of the letter and say "*this doesn't look right,*" or "*I don't want so much space here on the left side.*" Someone who knows that there are several alternative formatting commands for obtaining different forms of paragraphs will recognize from the printout that, for example, blocking them rather than indenting them would make it "look better."

There are two points I want to draw from this example: (a) There is a concrete result in the form of a visual representation to focus on; and (b) There are some objective or external criteria for solving the problem. In this case the user doesn't have to specify in any great detail what is wanted, because there are some objective standards for "neat" as opposed to "messy" letters.

I am contrasting the case of going to a travel agent and saying "I want a good holiday" (cf. "*how do I format a letter?*") with "I want to go somewhere less touristy than Mallorca" (cf. "*how do I make this look better?*") In the second case, both the travel agent and the person consulted about the letter start out knowing primarily what is not wanted. This is precisely the approach taken by Rabbit (see Figure 18.2), where



BEST COPY AVAILABLE

the interaction is arranged around presenting examples, which may then be criticized—i.e., you can say what you do or don't like about it. In other words, if the user's question is generated from some external representation, the designer can use "*answers first, then questions*" as a technique, within the basic question-answering paradigm, in order to circumvent the need for the user to know how to formulate questions.

Summary:

1. Questions may be driven by "internal" factors, such as goals and plans. There is a direct parallel here with the kind of analysis given by Moran's "external-to-internal task mapping" model (Moran, 1983)—the formulation of specific intentions from general goals and plans is a complex process, involving several stages. Support is needed for questions that arise at each of these stages. One approach to providing help for "internally-generated" questions, especially when users don't know what questions to ask, is to present an overview of the information, but as I have argued earlier, such an approach is undermined by the fact that users' tasks cut across the boundaries formed by any particular method of organizing the information.
2. Questions may also be driven by external objects or events in interacting with the system. Making things visible helps the user identify problems, but it can also help the system identify how it can help the user. The use of examples in the Rabbit system, Figure 18.2 capitalizes on this. What is needed is the ability for both the user and the system to share the *same* representation of the problem. For example, it would be nice if the user could point to an area of a soft-formatted version of their file, say something about it, and have the system know that what they are pointing to refers to an area in the input file, so that it can highlight where the problem is located, a point discussed by Draper in Chapter 16 (see Figure 18.5).

DIFFERENT TYPES OF QUESTIONS REQUIRE DIFFERENT TYPES OF DELIVERY SYSTEMS

In the course of trying to go from an initial vague question to the final answer—that is, in the process of carrying out planning supported by questions—various different types of questions can crop up, and should



14

ideally be answerable by the system. Let's go back to the example of a user asking *"how do I get a formatted printout of a letter?"* (see Figure 18.3). This general level question involves needing to ask several more specific questions because of the subtasks involved: For example, *"how do I get a letter heading?"*; *"how do I get the date printed on the right hand side?"*; *"how do I indent the paragraphs?"* etc. These questions are all of the same general type—that is, *"how?"* questions, which are handled more or less adequately by systems that can give some assistance on procedures for performing a task. However, in order to obtain the required information the user may also have to ask several different types of question: For example, *"how do I indent paragraphs?"* *"where do I place the .pp?"* *"should it be on a separate line?"* *"what if I didn't want the paragraph indented?"* etc.

Thus, there are often cases where the mapping, and therefore the translation process that has to go on, is one-to-many, and where the initial question leads to many *different* questions being generated. Furthermore, the answers to these questions may not be available immediately, but may have to be put on hold until the answers to intermediate questions, or solutions to subproblems, are found.

There are some types of question that are reasonably well supported by traditional designs such as menu-based help, command language or keyword systems, and so on. However, there are other kinds of questions that are not possible to ask, let alone answer, with traditional help systems.

There are several possible reasons for users needing to ask a question. They may have a goal to accomplish but don't know how to go about achieving it, or they may need a description of a term or concept. However, they also may just need confirmation or verification of a solution they are considering, or help in testing a hypothesis. They may also be considering several alternative solutions and need help in choosing among them.

What is the difference? Some questions are fairly straightforward, such as *"what is grep?"* The user needs a description of the command, which can be very brief or may involve a longer exposition especially if it is a concept they are asking about (e.g., a buffer) rather than a command. Although most systems are able to provide descriptive help of this kind more or less effectively, there is a variant of the *"what is?"* question that is often not taken into account: The question of the form *"what is the difference between...?"* Here the user needs to know the difference, or the relationship, between one or more things.

Suppose I'm trying to work out how to format a table of figures, and, during the course of getting help (imagine I'm browsing through

an online manual entry for *tbl*), I notice that there are some options available called "*box*" and "*allbox*." What I would like to be able to ask is "what is the difference between *box* and *allbox*?" I can imagine a system similar to the "Movie Manual" (Backer & Gano, 1983), that would allow me to call up visual examples of each. Figure 18.6 shows the kind of thing I mean. The manual entry contains highlighted words that are entities or concepts for which I can get examples or illustrations. There is a pop-up menu to the side of the window that contains several kinds of questions I can ask, including "*what is the difference?*" I click the mouse on the items I want illustrated, select the question type, and a window appears showing me the difference between *box* and *allbox* visually (see Figure 18.6B).⁴

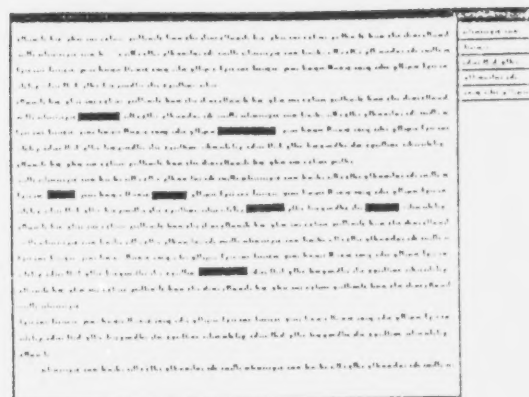
What if? Another common type of question that help systems often do address is the "*how?*" question—that is, requests for information about *how* to accomplish some goal. Most systems, however, assume that the user has specified this question well enough for the system to produce an immediate solution in the form of a procedure to carry out. In many cases, the "*how?*" questions of users may require further specification, which means that they will need some help in planning how to accomplish their higher level goal.

"*What if?*" questions are a special case of "*how?*" questions. These are questions about hypothetical situations, rather than being requests for recipes for performing some task. The answer to this type of question requires that the system be able to simulate the hypothetical case in some way.⁵

To extend the example, I want to be able to decide which of the two options (*box* and *allbox*) is most appropriate for my particular data, since my data are apt to look quite different from the examples given by the help system. Imagine I could point to my data file, then select the menu item "*what if?*" on the window containing the manual entry for *tbl* (see Figure 18.7), then point to the highlighted item "*box*," and a window opens up showing me a formatted version of my data file with that option set (see Figure 18.7B). This is a case of the system allowing me to simulate a hypothetical case in order to decide whether or not it's really what I want. (It is also helpful if, having decided it is what I want, I can go ahead and tell the system to make the relevant changes

⁴ Note that even if the manual entry told me that *box* means "enclose the table in a box," and that *allbox* means "enclose each item in a box," it is still much easier and quicker to look at the sets of contrasting examples shown in Figure 18.6.

⁵ Cf. Coombs and Alty (1984) and Rich (1982) for similar suggestions.



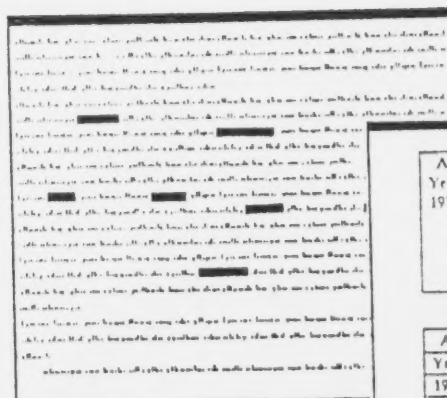
A



CONSULE



B



AT&T Common Stock		
Year	Price	Dividend
1971	41.54	\$2.60
2	41.54	\$2.70
3	46.55	\$2.87
4	40.53	\$3.24
5	45.52	\$3.40

box

AT&T Common Stock		
Year	Price	Dividend
1971	41.54	\$2.60
2	41.54	\$2.70
3	46.55	\$2.87
4	40.53	\$3.24
5	45.52	\$3.40

allbox



CONSULE



FIGURE 18.6. "What is the difference...?" (A) the manual entry contains highlighted words that are items for which the user can get examples or illustrations. The pop-up menu on the side of the window contains several different questions that can be asked—including "what is the difference?" The user clicks a mouse on the items to be illustrated, selects the question type, and a window appears (B) showing the difference between the items visually. In this case, window (B) answers the question, "What's the difference between the tbl formats box and allbox?"

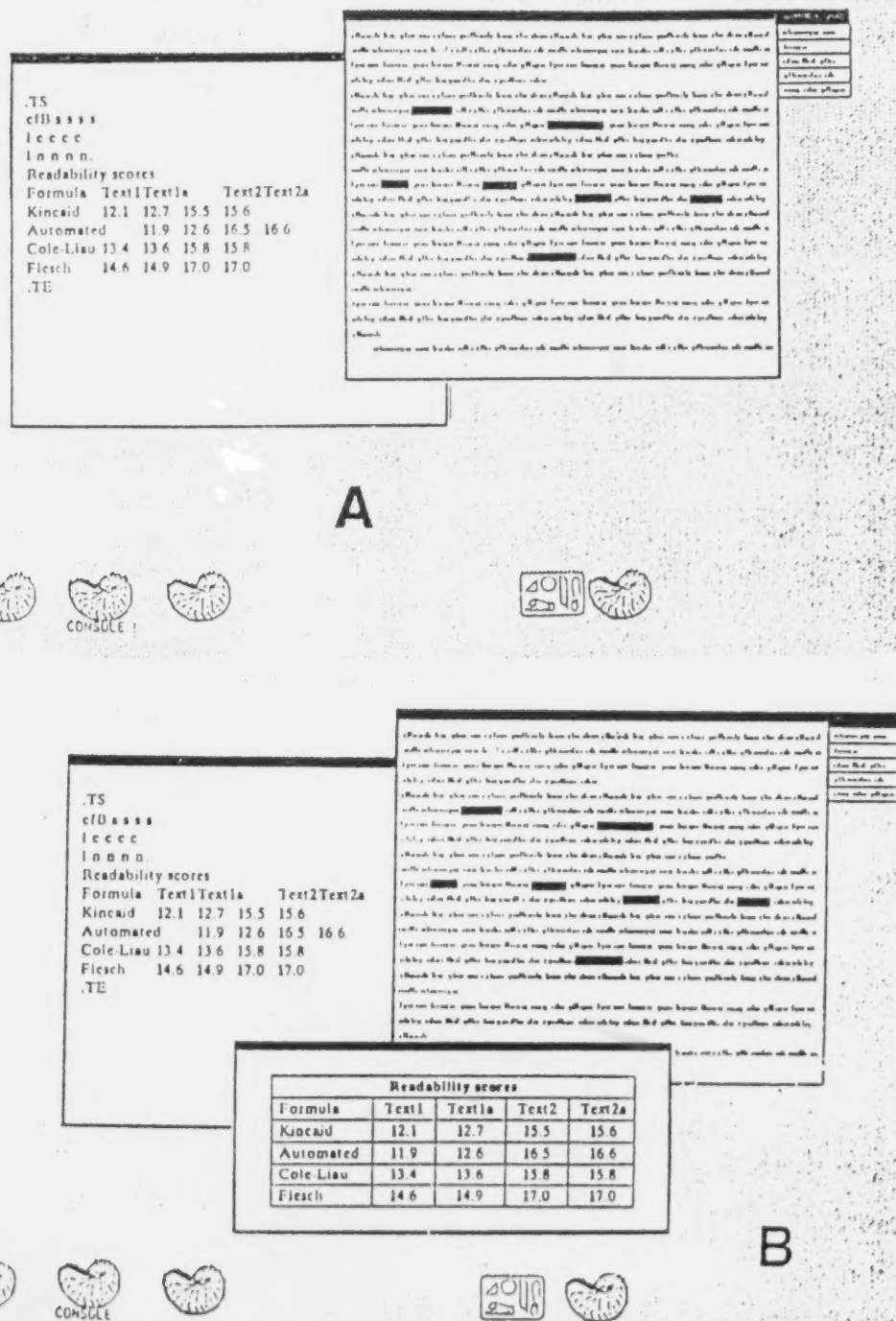


FIGURE 18.7. "What if?" (A) I am examining a manual entry, and I need to know how setting the option *box* would affect a formatted table of my particular data. I point to my data file, then select the menu item *what if*, then point to the highlighted item *allbox*, and a window opens (B), showing me a formatted version of my data file using the *allbox* format.

in my file. This sort of facility does exist, in abbreviated capability on commercial integrated spreadsheet and graphing systems such as Lotus, Symphony, Chart, Interleaf, etc.)

Why? These types of questions are usually interpreted as being a form of the question "*what caused...?*" and are viewed as requiring explanations based on the system providing a trace of the steps it went through to produce some result. Other interpretations of "*why?*" questions involve wanting to know why an undesirable result occurred. In other words, there is a difference between the question (a) "*where did I go wrong?*" and (b) "*why is this wrong?*"

Where did I go wrong? A simple way of supporting users in trying to work out why something went wrong is to show them where to look, without necessarily giving any further explanation. In other words, if the problem is due to a simple slip, the user can figure out what to do once it is pointed out. For example, let's say I've now decided what I need to put in my file in order to produce a formatted table of the data, and I go ahead and type the command "*tbl myfile | lnr.*" I then get the message "*tbl quits: error on line 19.*" I look at line 19 and see that I made a typographical error. Indicating where to look is all I need—I now know how to correct it. However, suppose the system went ahead and interpreted my command without any problem, but when I look at the output I don't like it for some reason. In this case, I want to know where the effect that I don't like originated. I would like to be able to point to the area that is wrong and have the system somehow tell me what had caused the failure, or at least, what alternatives I could have tried. The only difference is in whether the system or the user detected the existence of a problem; in both cases, a simple backtrace, to the origin of the problem without any deeper explanation, would suffice.

Why is this wrong? There are other cases however, where even when the source of the problem has been pointed out, the user still doesn't understand why something is wrong—perhaps because of some fundamental misconception. The distinction I'm making here with respect to "*why?*" questions is related to the distinction made between "slips" and "mistakes" (see Chapter 20 by Lewis & Norman). Slips (or questions of the type "*where did I go wrong?*") can be handled by drawing the user's attention to the location of the error. Mistakes (or questions of the type "*why is this wrong?*") are cases where there is a "bug" in the user's plan or model (i.e., "in the head"). In these cases some "deeper" explanation is required as the erroneous action may still "look right" when pointed to as the source of the problem.

THE NEED FOR INTEGRATED INFORMATION DELIVERY SYSTEMS

As I noted in the beginning of this discussion, users often go to other people for help rather than to manuals or help systems. I argued that this is not simply because they can get answers to their questions. For one thing, the answer that's given is often not a solution that the user can immediately implement, but rather some advice on where to find information from the documentation or help system. For another, the questions users ask are often initially stated at a very general level, so a good deal of dialogue has to go on in order to get to a specific enough level. Furthermore, each stage in this dialogue can lead to many different questions being generated.

The translation that is necessary in order to move from the first general question to the final answer puts additional loads on an already complex planning process. It would be disruptive if the user had to go to a different person for each step in the process of getting a solution to a problem, or for each different question. The apparent preference on the part of users for getting help from other people may be due to a need to have all the steps in the translation process supported, preferably from a single source. Thus, although I argue that several different methods of getting help should be made available to the user, it is important that all of these systems form a well-integrated whole. There should be no gaps in the system. That is, the help system ought to be able to cover all the possible types of query, as well as being able to provide the needed information. The user should also be able to go from each subsystem to the next in a smooth and uninterrupted fashion, so that the "chain of dialogue" is maintained.

Supporting self-documentation. There is another aspect to the point about the importance of maintaining the chain of dialogue in getting help: One thing that is not usually supported by help systems is the ability, once some piece of information has been found, to refind it at a later time, rather than go through the same process of problem solving over again. This is especially important when you come across information serendipitously, e.g., by browsing through a manual. You should be able to put some sort of bookmark or "dogear" in at that place. (A similar suggestion is made by Owen, Chapter 17.) People tend to do this for themselves anyway, in effect creating their own documentation: paper clips in their manuals, notes stuck on their terminals, little black books of useful gems, and so on.

There are two aspects to the issue of self-documentation. The first

is the need for *dynamic structures*—users should be able to structure or restructure the information to suit their own unique purposes. When users make notes from documentation, they filter out irrelevant items, mark important things to remember, and put the information in a unique perspective—one that it is unrealistic to expect the designer to anticipate. The second aspect concerns the need for *refinding information*—users should be able to refind useful information without having to go through the initial process of constructing the query again from scratch. With an online help system there should be some way of recording the use made of help that might allow the system to retrieve information that was previously accessed.⁶

One method of implementing these ideas is through the use of "hypertext." *Hypertext* is a term coined by Nelson (1967, 1981) to capture the notion of nonlinear or nonsequential text that allows for interactive branching and dynamic display of information. In a hypertext system a user can construct arbitrary links from any point in a document to other points within the same document or in another document. By using the resulting sets of links as "access structures," the user can retrieve information in a dynamically organized fashion. For example, a simple form of hypertext for this book might consist of a table of contents with links to files containing each chapter, each of which contains links to sections. As a reader, you could combine these links in any way you choose to construct your own organization of the text. Links can also replace footnotes to point to the actual material referenced.⁷

With such a system users can create links to new files in which they can comment on the information, so that the next time they access the information they also get back their own annotations. Such a system could also maintain histories or backtraces of the interaction via sets of links connecting the various modules of information. Reactivating these links by running a backtrace would automatically retrieve the modules pointed to by the links. These links would thus provide filters through which to view the information.⁸

⁶ Although the kind of "self-documentation" I discuss here concerns a single user being able to personalize the help system, the same point applies to whole groups of users—see the discussion of the DYK system in Chapter 17, by Owen.

⁷ Cf. Meyrowitz and Van Dam (1982).

⁸ See Feiner, Nagy, and Van Dam (1982) and Price (1982) for examples of systems that use such hypertext ideas.

SUMMARY AND CONCLUSIONS

In this chapter I have discussed some problems concerning user-initiated help—situations where users actively seek information to assist them with their tasks. Current help systems seem to employ four main methods in general for handling this:

1. Systems that answer "*what is?*" questions, but require knowledge of a particular terminology for accessing the information they contain—e.g., command-language help systems such as "*help*" in VMS, "*man*" in UNIX, and many database query systems.
2. Systems that are more flexible about the specification language—e.g., synonym-based keyword facilities, such as "*apropos*" in UNIX; natural language interfaces, or "quasi-natural language" systems (see Mishra, Trojan, Burke, & Douglas, 1984).
3. "Task-based" systems—e.g., systems that provide information according to users' tasks and goals, rather than in terms of the system boundaries or modules. Such systems are able to maintain the dynamic features characteristic of help that is generated "on the fly" by modularizing the information units to a level that allows multiple perspectives and access routes to information, since the modules can be combined in a flexible manner (cf. COUSIN, Hayes & Szekeley, 1983).
4. Systems that bypass the need for formal "help" by making the objects in the user's task environment "visible."

In this chapter I point out that users need help in asking questions; that asking the right question is not simply a matter of using the right terminology to phrase the question. Systems that are flexible about the specification language, and some menu systems, serve to "patch" the problem of not knowing how to ask for help; systems that are modular and "task-based" serve to "patch" the problem of not knowing the items for which help is available. However, these approaches still assume that users are only one step away from the information they need. They generally allow a very limited range of types of query, even if the "terminology" is flexible. I argue that there is a big gap between the user's initial queries and the specific questions they need to construct to elicit the information needed. The issue is to know how to support the user in formulating the question, a process that generally involves multiple steps. It is not enough to produce only the information alone—the system should also support question discover in order to.

support the whole user activity of information acquisition—of users helping themselves.

There are two aspects to this process, one involving the formulation of "internally generated" questions, the other involving "externally generated" questions. Current systems exhibit a tendency toward handling the first kind of question with natural language or database-query kinds of question-answering systems, and the second kind with "direct manipulation" interfaces. We should consider using both types of information delivery mechanisms more systematically for both kinds of question. For instance, with externally generated queries it should be possible to point to a word in an error message and initiate a dialogue. Conversely, perhaps someone can think of a "direct manipulation" method for handling the internally generated questions beginning by supplying task-based help in addition to the tool-based experimentation that is the most obvious kind of learning supported by direct-manipulation systems.

ACKNOWLEDGMENTS

I thank Liam Bannon, Sondra Buffett, and Don Norman for comments and advice on various drafts of this chapter, and Steve Draper for feedback and discussion of the ideas.